**NodeJS: Intro to Node.js, Events and Streams**

# Challenge 1:

**Hello you**

Let's start with a simple Hello server.

Follow the tasks below to create a simple Node server that outputs a greeting.

* First, tell the response which status it should have (a successful status is 200).
* Next, write a message to the response body in the form of "Hello, this is <your name here>"
* To finish it up, tell the response to end so the client on the other side knows it has received all the data.

var http = require('http');

http.createServer(function(request, response) {

}).listen(8080);

# Challenge 2:

**Convert Blocking**

Here's some code that is reading a file off the file-system using the blocking function readFileSync.

Convert the code to be non-blocking using the readFile function instead.

* Start by changing the call from readFileSync() to readFile ().
* Next, add a callback method to the readFile() call. This method should accept **error** and **contents** parameter.
* To finish it up, remove the contents var declaration, and move the call to console.log() inside your callback.

var fs = require('fs');

var contents = fs.readFileSync('index.html');

console.log(contents);

# Challenge 3:

**Read File in Server**

Now that you know how to create an HTTP server and how to read a file off the file system in a non-blocking way, let's try to combine the two. Instead of just writing a string to the HTTP response, write the contents of index.html to the response instead.

* After response.writeHead(200), add a call to fs.readFile() that reads index.html asynchronously. Remember to pass a callback function, that accepts an error parameter, and a contents parameter.
* Now that you have the file contents, write it to the response.
* To finish up, end the response after the file contents have been written.

var http = require('http');

var fs = require('fs');

http.createServer(function(request, response) {

  response.writeHead(200);

  response.end();

}).listen(8080);

# Challenge 4:

**Writing Response Headers**

Up until now all we've been sending into the response.writeHead() function is the status code. However, it can take additional parameters.

* Consult the node documentation, and add a Content-Type of text/html to the response.

var http = require('http');

var fs = require('fs');

http.createServer(function(request, response) {

  response.writeHead(200);

  fs.readFile('index.html', function(err, contents) {

    response.write(contents);

    response.end();

  });

}).listen(8080);

# Challenge 5:

**Response End**

Our original Hello server can be shortened since the response.end() function optionally takes data as a parameter. Remove the response.write line altogether, and send the hello string as a parameter on the response.end function. This will send the data, and once finished add the end to the response.

* Instead of passing the content to response.write(), pass it to response.end().
* Now, remove the call to response.write().

# Events

# Challenge 1:

**Chat Emitter**

We're going to create a custom chat EventEmitter.

* Create a new EventEmitter object and assign it to a variable called 'chat'.
* Next, let's listen for the 'message' event on our new chat object. Remember to add a callback that accepts the message parameter.
* Log the message to the console using console.log().

var events = require('events');

var EventEmitter = events.EventEmitter;

# Challenge 2:

**Emitting Events**

Read the existing code below and modify it to emit events.

* On the chat object, emit the 'join' event and pass in a custom message as a string.
* Now emit the 'message' event on the chat object. Just like before, remember to pass in a custom message as a string.

var events = require('events');

var EventEmitter = events.EventEmitter;

var chat = new EventEmitter();

var users = [], chatlog = [];

chat.on('message', function(message) {

  chatlog.push(message);

});

chat.on('join', function(nickname) {

  users.push(nickname);

});

// Emit events here

# Challenge 3:

**Request Event**

As you know, refactor the HTTP server code to explicitly bind a callback to the 'request' event using the on function.

* Add an event listener on the server variable that listens to the request event. The event listener should take a callback function with two arguments, request and response.
* Move the logic for handling the request from the http.createServer() callback to your new 'request' event listener.
* Remove the original request callback.

var http = require('http');

var server = http.createServer();

server.on('request', function(request, response) {

  response.writeHead(200);

  response.write("Hello, this is dog");

  response.end();

});

server.listen(8080);

# Challenge 4:

**Listening Twice**

Who said you can only listen for an event once?

* Add a second 'request' handler to the HTTP server.
* From inside of the new handler, log the message "New request coming in..." using console.log().

var http = require('http');

var server = http.createServer();

server.on('request', function(request, response) {

  response.writeHead(200);

  response.write("Hello, this is dog");

  response.end();

});

server.listen(8080);

# Challenge 5:

**Listening for Close**

Like our parents always used to say, listening is more important than talking! Modify the server so that we know when it's closed down.

* Listen for the 'close' event on the server. The event listener should take a callback function that accepts no arguments.
* Inside the 'close' callback, log the message "Closing down the server..."

var http = require('http');

var server = http.createServer();

server.on('request', function(request, response) {

  response.writeHead(200);

  response.write("Hello, this is dog");

  response.end();

});

server.listen(8080);

# Streams

# Challenge 1:

**File Piping**

Instead of manually listening for the 'readable' event on the Readable stream, let's use pipe to read from the stream and write directly to process.stdout.

* Start by removing the code for the readable handler.
* Call file.pipe(), passing it the stream to write to.

var fs = require('fs');

var file = fs.createReadStream('fruits.txt');

file.on('readable', function(){

  var chunk;

  while(null !== (chunk = file.read())){

    console.log(chunk.toString());

  }

});

# Challenge 2:

**Fixing Pipe**

The following code will throw an error because pipe automatically closed our writable stream.

* You'll need to consult the pipe documentation to figure out the option which keeps the Write stream open and dispatches the end event.

var fs = require('fs');

var file = fs.createReadStream('origin.txt');

var destFile = fs.createWriteStream('destination.txt');

file.on('end', function () {

  destFile.end('Finished!');

});

# Challenge 3:

**Download Server**

Let's create an HTTP server that will serve index.html.

* Use pipe() to send index.html to the response.

var fs = require('fs');

var http = require('http');

http.createServer(function(request, response) {

  response.writeHead(200, {'Content-Type': 'text/html'});

  var file = fs.createReadStream('index.html');

}).listen(8080);